ParkLogic : Rapport technique (P.F.M.)

Decembre 29, 2025

Bilal Houari
houari.bilal@etu.uae.ac.ma
Groupe 10

Ayman Amtot
amtot.aymane@etu.uae.ac.ma
Groupe 10

Adil Elbahlouli
elbahlouli.adil@etu.uae.ac.ma
Groupe 10

Safouane Bousakhra
safouane.bousakhra@etu.uae.ac.ma
Groupe 10

Saad Marzouq
marzouq.saad@etu.uae.ac.ma
Groupe 10

Table des matières

I. Introduction

ParkLogic est un environnement de simulation développé en C++20, conçu pour modéliser et analyser les interactions entre les véhicules et les infrastructures au sein d'un environnement structuré. Ce projet sert de démonstration technique des principes de la programmation orientée objet (POO) et des mécaniques de base de simulation, en se concentrant spécifiquement sur la navigation d'agents autonomes, l'assemblage procédural de cartes (procedural map assembly) et la gestion des ressources (parkings et bornes de recharge).

I.1. Mission et portée

L'objectif principal de ParkLogic est de fournir un environnement déterministe pour simuler le comportement des véhicules à mesure que les agents interagissent avec divers modules d'infrastructure. Cela inclut :

- **Interaction véhicule-infrastructure** : Modélisation de la manière dont les voitures identifient les parkings et les infrastructures de recharge pour véhicules électriques (EV), comment elles y naviguent et comment elles les utilisent.
- **Génération procédurale de cartes** : Utilisation d'un système basé sur une grille (grid-based system) pour assembler des routes et des installations en fonction de paramètres de configuration définis par l'utilisateur.
- **Arbitrage de destination basé sur des heuristiques** : Implémentation de règles logiques pour gérer les choix de destination des véhicules en se basant sur la proximité et le prix.

Bien que conçu comme un projet académique, l'implémentation respecte plusieurs pratiques modernes d'ingénierie logicielle, notamment l'utilisation de sous-systèmes découplés, une communication pilotée par les événements (event-driven communication) et une logique de type fixed-timestep.

I.2. Stack technologique

Le développement de ParkLogic repose sur une sélection ciblée d'outils et de bibliothèques pour gérer le rendu, la logique et l'orchestration du build :

- **Langage principal** : C++20. Le projet exploite les fonctionnalités de la bibliothèque standard telles que `<format>` pour le logging, `<deque>` pour la gestion des waypoints, et `std::type_index` pour une gestion d'événements type-safe. Les smart pointers (`std::unique_ptr` , `std::shared_ptr`) sont utilisés tout au long du projet pour gérer les cycles de vie des objets.
- **Moteur de rendu** : **Raylib 5.5**. Raylib est utilisé pour le rendu 2D avec accélération matérielle et pour la gestion des fenêtres. Il assure la transition visuelle entre les unités de simulation internes et l'écran de l'utilisateur.
- **Intégration de la logique** : La simulation fonctionne sur une **intégration à pas de temps fixe (Fixed-Timestep) de 60Hz**. Cela garantit que les calculs d'IA et de physique restent déterministes et cohérents, quel que soit le framerate de rendu.
- **Framework de tests** : **GoogleTest (GTest)**. Des tests unitaires (unit tests) sont intégrés via CMake pour vérifier les composants essentiels tels que l' `EventBus` , la `GameLoop` et le `SceneManager` .
- **Système de build** : **CMake 3.20+**. La gestion des dépendances est automatisée à l'aide de `FetchContent` , garantissant un environnement cohérent pour la compilation du projet et de ses bibliothèques externes (Raylib et EnTT).

II. Aperçu général et architecture système

Les sections suivantes détaillent l'organisation structurelle du projet ainsi que l'implémentation technique de ses sous-systèmes principaux.

> *Remarque : Les descriptions de système fournies dans cette section sont des simplifications de haut niveau destinées à établir le contexte. Les spécifications techniques exhaustives pour chaque module et ses mécaniques internes sont fournies dans Chapitre III.*

II.1. Structure technique des fichiers

Le projet est organisé en domaines logiques, séparant les utilitaires du moteur principal des entités et systèmes spécifiques à la simulation. L'arborescence suivante présente la structure principale du répertoire `include/` :

```
include/
├── config.hpp                # Paramétrage central et constantes
├── core/
│   ├── Application.hpp       # Coordinateur global
│   ├── AssetManager.hpp      # Mise en cache des ressources
│   ├── AudioManager.hpp      # Musique et effets sonores globaux
│   ├── EntityManager.hpp     # Cycle de vie des entités et ordre de rendu
│   ├── EventBus.hpp          # Système de publication/abonnement (Pub/Sub)
│   ├── EventLogger.hpp       # Écouteur de diagnostics
│   ├── GameLoop.hpp          # Gestion du temps et synchronisation
│   ├── Logger.hpp            # Abstraction de la console (journalisation)
│   └── Window.hpp            # Gestion du DPI et de la résolution
├── entities/
│   ├── Car.hpp               # Logique de l'agent et physique
│   ├── Entity.hpp            # Type de base polymorphe
│   └── map/
│       ├── Modules.hpp       # Tuiles de grille (Routes/Infrastructures)
│       ├── Waypoint.hpp      # Marqueurs de chemin
│       ├── WorldGenerator.hpp # Logique d'assemblage procédural
│       └── World.hpp         # Conteneur de la carte et rendu de la grille
├── events/
│   ├── EventTypes.hpp        # Définitions des types d'événements de base
│   ├── GameEvents.hpp        # Déclencheurs spécifiques à la simulation
│   ├── InputEvents.hpp       # Pont pour le clavier et la souris
│   ├── TrackingEvents.hpp    # Déclencheurs de suivi de la vue (viewport)
│   └── WindowEvents.hpp      # Déclencheurs système/fenêtre
├── input/
│   └── InputSystem.hpp       # Récupération (polling) et publication de l'état matériel
├── scenes/
│   ├── GameScene.hpp         # Scène principale de la simulation
│   ├── IScene.hpp            # Interface pour la machine à états
│   ├── MainMenuScene.hpp     # Interface d'accueil de l'application
│   ├── MapConfigScene.hpp    # Écran des paramètres de pré-simulation
│   └── SceneManager.hpp      # Logique de transition entre scènes
├── systems/
│   ├── CameraSystem.hpp      # Manipulation de la vue (Zoom/Panoramique)
│   ├── PathPlanner.hpp       # Génération de trajectoires
│   ├── TrackingSystem.hpp    # Suivi automatique d'un agent
│   └── TrafficSystem.hpp     # Règles globales et orchestration des apparitions (spawn)
└── ui/
    ├── DashboardOverlay.hpp  # Surveillance en temps réel (tableau de bord)
    ├── GameHUD.hpp           # Barre de contrôle en simulation
    ├── UIButton.hpp          # Primitives interactives (boutons)
    ├── UIElement.hpp         # Classe de base de l'interface utilisateur
    └── UIManager.hpp         # Groupement de l'HUD et routage d'événements
```

II.2. Sous-systèmes du moteur principal (Core Engine)

La logique d'exécution principale est gérée par plusieurs composants centralisés :

- **Coordinateur de l'application (Application Coordinator)** : La classe `Application` fait office de propriétaire global de tous les sous-systèmes de haut niveau, gérant la boucle `run()` principale et orchestrant les procédures d'initialisation et de fermeture.
- **Communication pilotée par les événements (Event-Driven Communication)** : Un `EventBus` de type-safe facilite la communication entre des composants disparates. Cela permet une architecture découplée (decoupled architecture) où, par exemple, un bouton de l'interface utilisateur peut déclencher l'apparition (spawn) d'un véhicule sans avoir de dépendance directe avec le code de simulation du trafic.
- **Gestion des états (State Management)** : Le `SceneManager` gère les transitions entre les différents états de l'application, tels que la `MainMenuScene`, la `MapConfigScene` et la `GameScene` principale. Cela garantit un nettoyage de la mémoire (memory cleanup) maîtrisé lors des changements d'état.
- **Boucle d'exécution (Execution Loop)** : La `GameLoop` implémente un algorithme de type « Fix Your Timestep ». En utilisant un accumulateur, la simulation maintient son taux de mise à jour logique de 60Hz tout en permettant des vitesses de rendu (rendering speeds) variables.
- **Gestion des ressources** : Un `AssetManager` assure le chargement différé (*lazy loading*) et la mise en cache des textures, des sons et des flux musicaux, réduisant ainsi les allocations mémoire redondantes et centralisant la gestion des ressources.
- **Services audio** : Un `AudioManager` dédié gère la lecture de la musique d'ambiance, les effets sonores globaux et l'état de l'interface de sourdine unifiée.

II.3. Infrastructure de la carte et de la simulation

L'environnement physique est représenté à travers un système modulaire basé sur une grille :

- **Génération du monde (World Generation)** : Le `WorldGenerator` valide et place les segments de route et les installations en fonction de règles de connectivité logique. Les points d'attache corrects et les décalages (offsets) des jonctions en T sont calculés lors de la phase de génération.
- **Normalisation des coordonnées** : La simulation fait la distinction entre les **Mètres** internes (utilisés pour l'IA et la physique) et les **Pixels artistiques** (utilisés pour le rendu). La conversion est gérée via une constante définie de Pixels-Par-Mètre (PPM), assurant une cohérence spatiale sur différentes résolutions d'affichage.
- **Hiérarchie des modules** : L'infrastructure est catégorisée en segments de type `Road` pour le transit et en installations spécialisées `Parking/Charging Facilities`. Ces modules contiennent des entités de type `Spot` discrètes avec des états associés (Libre, Réservé, Occupé) et des métadonnées économiques (multiplicateurs de prix).

II.4. Simulation du trafic et IA des agents

Les véhicules opérant au sein de l'environnement sont gérés via un coordinateur macroscopique et une logique d'agent individuelle :

- **Orchestration du trafic** : Le `TrafficSystem` gère l'apparition (spawn) des véhicules en fonction des niveaux définis par l'utilisateur et surveille les taux d'occupation globaux.
- **Heuristiques de destination** : Lorsqu'un véhicule entre dans la simulation, il sélectionne une destination en fonction de stratégies de priorité configurables, telles que le prix le plus bas ou la distance physique la plus courte.
- **Machine à états de l'IA (AI State Machine)** : Les agents passent par différents états incluant `DRIVING` (transit sur la route principale), `ALIGNING` (manœuvre vers les portes de l'installation), `PARKED` (recharge stationnaire ou sessions d'attente) et `EXITING` (retour vers le réseau routier).
- **Direction et navigation (Steering and Navigation)** : Le mouvement est régi par des comportements de direction (Steering behaviors : Seek et Arrival). Un `PathPlanner` génère des trajectoires géométriques en assemblant des séquences de `Waypoint` basées sur les décalages de voies (lane offsets) et les configurations spécifiques aux installations.

III. Spécifications détaillées des composants

Cette section fournit une analyse technique semi-exhaustive de la base de code de ParkLogic. Chaque composant est examiné sous l'angle de sa responsabilité, de son état interne et de son interaction avec les autres sous-systèmes.

III.1. Core Engine & Infrastructure de base

La couche du moteur principal (Core Engine) fournit les services essentiels nécessaires au fonctionnement de la simulation, notamment la temporisation, le routage d'événements et la gestion des ressources.

III.1.a. Classe `Application`

La classe `Application` (`Application.hpp` / `Application.cpp`) est l'autorité centrale et le point d'entrée du logiciel. Elle agit comme racine de composition (Composition Root) : elle assure la propriété (ownership) et le cycle de vie des composants fondamentaux, tout en déléguant la logique métier à ses membres spécialisés.

Responsabilités :

- **Orchestration du système** : Initialise et possède le cycle de vie de tous les sous-systèmes primaires : `EventBus` , `Window` , `InputSystem` , `SceneManager` , `EventLogger` et `GameLoop` .
- **Gestion globale des événements** : S'abonne aux événements système de haut niveau qui affectent l'état global de l'application (par exemple, arrêter la boucle lorsque la fenêtre est fermée).
- **Cycle de vie des ressources** : Gère le chargement des assets globaux (via l' `AssetManager`) et l'initialisation/ fermeture des périphériques matériels comme le périphérique audio.
- **Exécution de haut niveau** : Fournit la méthode `run()` , qui lance la `GameLoop` .

Détails d'implémentation technique :

- **Intégration audio** : L' `Application` délègue toute la gestion audio à l' `AudioManager` . Elle initialise le périphérique audio matériel au démarrage et assure sa fermeture sécurisée. La musique d'ambiance et l'interface de sourdine globale sont gérées en tant que services autonomes du moteur.- **Abonnements aux événements** :
  - `WindowCloseEvent` : Bascule le flag `isRunning` à `false` , provoquant la terminaison propre de la boucle de jeu.
  - `SimulationSpeedChangedEvent` : Transmet les multiplicateurs de vitesse de l'interface utilisateur vers la `GameLoop` .
  - `SceneChangeEvent` : Réinitialise le multiplicateur de vitesse de simulation à 1.0 si l'utilisateur quitte la scène de simulation.
- **Intégration de la boucle principale** : La méthode `run()` contient les définitions lambda pour les étapes `update` et `render` , qui sont passées à la `GameLoop` pour exécution à intervalle fixe.
- **Ordre d'affichage (Draw Ordering)** : La méthode `render()` coordonne l'ordre d'affichage de base : vérification de l'état de fermeture de la fenêtre, mise à jour des entrées (*inputs*), initialisation du contexte de rendu, affichage de la scène actuelle et délégation du rendu de l'interface utilisateur globale à l' `AudioManager` .

Exemple de code : Propriété et Initialisation

```cpp
// Application agit comme le propriétaire central des sous-systèmes du core
eventBus = std::make_shared<EventBus>();
window = std::make_unique<Window>(eventBus);
inputSystem = std::make_unique<InputSystem>(eventBus, *window);
sceneManager = std::make_unique<SceneManager>(eventBus);
eventLogger = std::make_unique<EventLogger>(eventBus);
gameLoop = std::make_unique<GameLoop>();
```

III.1.b. Classe `GameLoop`

La classe `GameLoop` (`GameLoop.hpp` / `GameLoop.cpp`) implémente la logique temporelle centrale de la simulation, garantissant que les mises à jour logiques sont déterministes, quelles que soient les performances de rendu du matériel.

Responsabilités :

- **Intégration temporelle déterministe** : Implémente l'algorithme « **Fix Your Timestep** » pour découpler la fréquence de simulation de la fréquence de rendu.
- **Modulation de la vitesse de simulation** : Gère un `speedMultiplier` qui permet à l'utilisateur d'accélérer ou de ralentir l'écoulement du temps dans la simulation sans affecter la stabilité des calculs physiques.
- **Contrôle du flux** : Exécute les lambdas d'update et de render fournis à l'intérieur d'une boucle `while` qui surveille un état « running ».

Détails d'implémentation technique :

- **Fixed Delta Time** : La boucle utilise une constante codée en dur `Config::FIXED_DELTA_TIME` (généralement fixée à 1/60ème de seconde).
- **Accumulateur temporel** : Elle utilise un `double accumulator` pour suivre le temps écoulé. À chaque frame, elle calcule le `frameTime` (plafonné à 0,25 seconde pour éviter la « **spiral of death** » lors de lags importants), le multiplie par le `speedMultiplier`, et l'ajoute à l'accumulateur.
- **Modèle de consommation** : Une boucle `while` interne consomme le temps de l'accumulateur par tranches de `dt`, appelant la fonction `update(dt)` pour chaque tranche. Cela garantit que si une frame prend 1/30ème de seconde, la mise à jour logique est appelée exactement deux fois.
- **Planification du rendu (Render Scheduling)** : La fonction `render()` est appelée exactement une fois après que toutes les mises à jour logiques en attente pour la frame actuelle ont été traitées.

Exemple de code : Logique de Fixed Timestep

```cpp
const double dt = Config::FIXED_DELTA_TIME;
while (running()) {
    accumulator += (GetTime() - currentTime) * speedMultiplier;
    while (accumulator >= dt) {
        update(dt);
        accumulator -= dt;
    }
    render();
}
```

III.1.c. Système `EventBus`

L'`EventBus` (`EventBus.hpp`) est la colonne vertébrale de l'architecture de communication du projet. Il s'agit d'une implémentation **header-only**, **type-safe** et **thread-safe** du **Design Pattern Publish-Subscribe**.

Responsabilités :

- **Découplage des composants** : Permet à différentes parties de l'application (par exemple, `TrafficSystem`, `UIManager`, `Car`) de communiquer sans inclusions directes de headers ou de dépendances d'objets.
- **Dispatching Type-Safe** : Utilise le RTTI de C++ (`std::type_index`) pour garantir que les écouteurs ne reçoivent que les événements du type spécifique auquel ils se sont abonnés.
- **Gestion du cycle de vie des abonnements** : Fournit un token d'abonnement **RAII** (`Subscription`) qui désabonne automatiquement l'écouteur lorsqu'il est détruit.

Détails d'implémentation technique :

- **Stockage interne** : Les abonnés sont stockés dans une `std::unordered_map` où la clé est un `std::type_index` et la valeur est un `std::vector` de pointeurs `IEventWrapper` dont le type a été effacé (**Type Erasure**).
- **Modèle de Thread Safety** :
  - Utilise un `std::shared_mutex` pour permettre des appels `publish` concurrents (**Shared Lock**).
  - Utilise un **Unique Lock** pour les opérations `subscribe` et `unsubscribe`.
- **Exécution par Snapshot** : Pour supporter la réentrance (par exemple, un callback se désabonnant lui-même pendant l'exécution), la méthode `publish` prend une copie locale (**snapshot**) de la liste des abonnés tout en maintenant le verrou, puis exécute les callbacks en dehors du verrou.

- **Type Erasure** : Le template `EventWrapper<T>` hérite d'un `IEventWrapper` non-templatisé, permettant au bus de stocker des types de callbacks disparates dans le même conteneur tout en conservant la capacité de faire un cast vers le type d'origine lors du dispatch.

Exemple de code : Abonnement et Publication

```cpp
// Les composants s'abonnent aux événements via des tokens RAII
auto token = eventBus->subscribe<CarSpawnedEvent>([](const auto& e) {
    Logger::Info("Voiture créée à : {}", e.car->getPosition());
});

// N'importe quel système peut publier des événements sur le bus global
eventBus->publish(CarSpawnedEvent{ newCarPtr });
```

III.1.d. Classe `SceneManager`

Le `SceneManager` (`SceneManager.hpp` / `SceneManager.cpp`) gère la machine à états de haut niveau de l'application, s'occupant de la permutation et du cycle de vie des différentes « Scènes » de jeu.

Responsabilités :

- **Transitions d'état** : Fournit des méthodes pour basculer entre les scènes `MainMenu` , `MapConfig` et `Game` .
- **Gestion de la propriété** : Détient le `std::unique_ptr` de la scène actuellement active, garantissant que la scène précédente est correctement désallouée lors d'une transition.
- **Délégation Update/Render** : Transmet les appels de haut niveau `update` et `render` de l' `Application` vers la scène active.

Détails d'implémentation technique :

- **Design Pattern Factory** : La méthode `setScene(SceneType type)` contient la logique pour instancier les classes de scènes concrètes.
- **Injection de dépendances** : Transmet le `shared_ptr<EventBus>` à chaque scène qu'il crée, garantissant qu'elles peuvent communiquer avec le reste de l'application.
- **Logique de transition** : Lorsqu'une nouvelle scène est définie, il publie un `SceneChangeEvent` sur l' `EventBus` , permettant aux autres systèmes (comme l' `Application` ou la `GameLoop`) de réagir au changement.

III.1.e. Classe `EntityManager`

L' `EntityManager` (`EntityManager.hpp` / `EntityManager.cpp`) agit comme le registre et le coordinateur de toutes les entités spécifiques à la simulation, incluant la carte du monde, les modules d'infrastructure et les véhicules.

Responsabilités :

- **Coordination du cycle de vie des entités** : Gère la création, le stockage et la destruction des objets `World` , `Module` et `Car` .
- **Assemblage piloté par les événements** : Génère le monde et ses modules en réponse à un `GenerateWorldEvent` .
- **Rendu par Z-Order** : Garantit que les entités sont dessinées dans le bon ordre spatial (Arrière-plan du monde -> Modules -> Voitures -> Overlays de premier plan).
- **Synchronisation globale** : Transmet les signaux de mise à jour à toutes les entités gérées, s'assurant qu'elles restent synchronisées avec l'horloge de la simulation.

Détails d'implémentation technique :

- **Registre dynamique** : Stocke les entités en utilisant des `std::vector<std::unique_ptr<T>>` .
- **Apparition interactive (Spawning)** : S'abonne à `CreateCarEvent` pour instancier de nouveaux agents `Car` à des positions spécifiques avec des priorités désignées.
- **Feedback de simulation** : Publie `CarSpawnedEvent` et `WorldBoundsEvent` pour informer les autres systèmes de l'état de la simulation.
- **Logique de sélection** : Suit quelle `Car` est actuellement sélectionnée via l'interface utilisateur et gère la visibilité de la visualisation du chemin de débogage (`draw(showPath)`).

- **Nettoyage** : La méthode `clear()` réinitialise l'intégralité de l'état de la simulation, ce qui est typiquement déclenché avant la génération d'une nouvelle carte du monde.

III.1.f. Classe `AssetManager`

L'`AssetManager` (`AssetManager.hpp` / `AssetManager.cpp`) implémente le **Design Pattern Singleton** pour fournir un accès global aux textures et sons mis en cache, évitant ainsi les allocations redondantes dans la mémoire GPU.

Responsabilités :

- **Mise en cache centralisée** : Stocke les objets `Texture2D`, `Sound` et `Music` dans des registres internes `std::map`, indexés par des chaînes de caractères uniques.
- **Standardisation des ressources** : Implémente une méthode `LoadAllAssets()` qui sert de source unique de vérité pour toutes les ressources de la simulation, garantissant que les routes, les tuiles, les véhicules et les flux audio sont chargés une seule fois au démarrage.
- **Chargement différé et nettoyage manuel** : Garantit que les ressources ne sont chargées depuis le disque qu'une seule fois et fournit une méthode `UnloadAll()` robuste pour libérer la mémoire GPU et audio lors de la fermeture.

Détails d'implémentation technique :

- **Accès Singleton** : Contrôlé via une méthode statique `Get()` qui renvoie une référence à une instance locale statique.
- **Intégration Raylib** : Utilise les fonctions internes de Raylib `::LoadTexture()`, `::LoadSound()` et `::LoadMusicStream()`.
- **Gestion des erreurs** : Enregistre un avertissement (*warning*) si une ressource est déjà chargée et une erreur si le chemin du fichier est invalide ou corrompu.

III.1.g. Classe `AudioManager`

L'`AudioManager` (`AudioManager.hpp` / `AudioManager.cpp`) est un singleton de service dédié, responsable de l'environnement sonore de la simulation et de l'interface utilisateur audio globale.

Responsabilités :

- **Gestion des flux musicaux** : Gère la lecture, la mise à jour et la normalisation du volume de la musique d'ambiance.
- **Interface de sourdine globale** : Détient et gère la logique et le rendu du bouton `UIButton` de sourdine globale ainsi que ses icônes de volume associées.
- **Orchestration des effets sonores** : Fournit une interface centralisée pour la lecture d'effets sonores ponctuels (ex: clics de boutons) avec modulation optionnelle de la hauteur du son (*pitch*).

Détails d'implémentation technique :

- **Découplage de l'interface utilisateur** : En prenant la responsabilité de la logique du bouton de sourdine, l'`AudioManager` déleste la classe `Application`, déplaçant la gestion des événements UI et le positionnement des icônes dans un contexte audio spécialisé.
- **Intégration des ressources** : Récupère toutes les ressources audio directement auprès de l'`AssetManager`.
- **Contrôle de l'état global** : Gère l'indicateur `isMuted` (est sourd) et coordonne les modifications du volume principal pour l'ensemble du contexte Raylib.

III.1.h. Sous-système `Window`

La classe `Window` (`Window.hpp` / `Window.cpp`) gère le pont entre la fenêtre physique du système d'exploitation et le système de coordonnées logiques interne du moteur.

Responsabilités :

- **Indépendance de la résolution** : Maintient une résolution logique persistante (définie dans `Config::LOGICAL_WIDTH/HEIGHT`) indépendamment de la taille réelle de la fenêtre.
- **Correction du ratio d'aspect (Aspect Ratio)** : Implémente le **letterboxing/pillarboxing** pour garantir que la simulation ne s'étire pas sur des moniteurs aux ratios d'aspect différents.
- **Mise à l'échelle et décalages (Offsets)** : Calcule le facteur d'échelle (`scale`) et le décalage (`offset` X, Y) nécessaires pour centrer le cadre logique dans la fenêtre physique.

- **Dispatching d'événements** : Publie `WindowResizeEvent` pour informer les autres systèmes (comme l'UI ou la Caméra) lorsque les dimensions de l'écran changent.

Détails d'implémentation technique :

- **Render Texture (Letterboxing)** : La classe utilise une `RenderTexture2D` de Raylib comme **backbuffer**. Tous les appels de rendu sont initialement dirigés vers cette texture à la résolution logique.
- **Logique de mise à l'échelle** : La méthode `updateDimensions()` calcule l'échelle comme `std::min(screenWidth/logicalWidth, screenHeight/logicalHeight)`.
- **Projection** : Dans `endDrawing()`, la classe dessine la texture du backbuffer sur l'écran en utilisant `DrawTexturePro`, en appliquant l'échelle calculée et les décalages de centrage.
- **Filtrage** : Utilise un filtrage de texture bilinéaire pour garantir que l'image mise à l'échelle reste nette.

Logique mathématique de mise à l'échelle : Le système préserve le ratio d'aspect logique en trouvant le facteur d'échelle minimum qui s'adapte soit à la largeur, soit à la hauteur de l'écran, puis en calculant les décalages pour centrer le cadre résultant.

```cpp
// 1. Calculer le facteur d'échelle uniforme
scale = std::min((float)GetScreenWidth() / logicalWidth,
                 (float)GetScreenHeight() / logicalHeight);

// 2. Calculer les décalages pour le centrage (Letterboxing/Pillarboxing)
offsetX = (GetScreenWidth()  - (logicalWidth  * scale)) * 0.5f;
offsetY = (GetScreenHeight() - (logicalHeight * scale)) * 0.5f;
```

III.1.i. Utilitaires `Logger` et `EventLogger`

Le sous-système de logging fournit à la fois une interface console destinée au développeur et une trace de diagnostic pilotée par les événements.

`Logger` (Utilitaire statique) :

- **Responsabilités** : Fournit une sortie console thread-safe et catégorisée (`Info`, `Warn`, `Error`).
- **Détails techniques** : Utilise le module `<format>` de C++20 et des templates variadiques pour un formatage de message type-safe. Il utilise un `std::mutex` pour empêcher l'entrelacement des sorties provenant de plusieurs threads (par exemple, le thread de simulation par rapport au thread de rendu).

`EventLogger` (Écouteur de diagnostic) :

- **Responsabilités** : S'abonne à l'`EventBus` pour créer un récit en direct des activités de la simulation (par exemple, « Voiture apparue en [X, Y] », « Scène changée en Game »).
- **Implémentation des assets** : Il détient une collection de tokens `Subscription` pour chaque type d'événement qu'il surveille. Lorsqu'un événement survient, il formate les données de l'événement en une chaîne lisible par l'homme et la transmet au `Logger` de base.
- **Cas d'utilisation** : Essentiel pour déboguer des interactions complexes où le feedback visuel est insuffisant (par exemple, vérifier qu'une voiture a correctement réservé une place avant son arrivée).

III.2. Infrastructure de la carte et du monde

Cette section couvre la génération procédurale et les composants physiques de l'environnement de simulation.

III.2.a. Classe `WorldGenerator`

Le `WorldGenerator` (`WorldGenerator.hpp` / `WorldGenerator.cpp`) est un moteur de placement procédural qui assemble la carte de simulation à partir de modules discrets en fonction d'une configuration cible.

Responsabilités :

- **Planification déterministe de la carte** : Calcule la séquence de routes et d'installations nécessaires pour satisfaire la demande de l'utilisateur (par exemple, « 5 stations de recharge »).

- **Alignement géométrique** : Gère l'assemblage linéaire (**linear stitching**) des modules en utilisant des points d'attache (**attachment points**) et des vecteurs normaux.
- **Évitement de collision** : Implémente un **lookahead « Safe-X »** pour s'assurer que les installations attachées aux routes suivantes ne chevauchent pas les infrastructures précédentes.
- **Finalisation du monde** : Calcule les limites globales du monde, les aligne sur des multiples de tuiles d'arrière-plan et normalise toutes les coordonnées afin que la route principale commence à X=0. L'allocation des ressources pour les tuiles du monde est gérée exclusivement par la routine de préchargement de l' `AssetManager`.

Détails d'implémentation technique :

- **Génération en deux phases** :
  1. **Le Plan (The Plan)** : Détermine quel type de route (Up, Down ou Double entrance) est nécessaire pour chaque installation et assigne les installations à ces routes en tant qu'enfants.
  2. **Le Placement (The Placement)** : Parcourt le plan, place les modules de route et calcule la position mondiale des enfants. Si la largeur d'une installation risque de provoquer une collision avec une installation précédente, le générateur insère des segments de type « **Normal Road** » (padding) jusqu'à ce qu'une distance de sécurité soit atteinte.
- **Logique de normalisation** : Après le placement, le générateur calcule les limites minimales et maximales. Il applique un décalage (offset) à tous les modules pour qu'ils soient centrés verticalement et commencent à l'abscisse X=0.
- **Continuité visuelle** : Ajoute des « **External Roads** » à l'extérieur des limites du monde pour donner l'impression que les véhicules arrivent et repartent vers un réseau infini.

L'algorithme de génération en deux phases 1. **Le Plan** : Détermine la séquence des modules en fonction du nombre d'installations cibles. Il attribue un type « **Road Entrance** » (Up, Down ou Double) à chaque segment de route selon les besoins. 2. **Le Placement** : Calcule itérativement la `worldPosition` de chaque route. Pour chaque route, il calcule les positions globales de ses installations attachées. Si une collision est prédite à l'aide d'un **lookahead**, il insère des segments de remplissage de type « **Normal Road** ».

III.2.b. Hiérarchie d'héritage des `Module`

La classe de base `Module` et ses dérivés définissent chaque objet physique de la simulation avec lequel les agents peuvent interagir.

Responsabilités :

- **Définition spatiale** : Définit les dimensions et la position mondiale en mètres.
- **Logique d'attachement** : Stocke des objets `AttachmentPoint` (position + normale) utilisés par le `WorldGenerator`.
- **Métadonnées de navigation** : Détient une collection d'objets `LocalWaypoint` qui définissent les chemins que les véhicules doivent suivre à l'intérieur du module.
- **État économique** : Suit un `priceMultiplier` et les prix individuels des places (`Spot`) pour les zones de parking ou de recharge.

Sous-classes principales :

- **Modules de route (Road Modules)** : `NormalRoad`, `UpEntranceRoad`, `DownEntranceRoad`, `DoubleEntranceRoad`. Ceux-ci définissent le chemin de transit principal.
- **Modules d'installation (Facility Modules)** : `SmallParking`, `LargeParking`, `SmallChargingStation`, `LargeChargingStation`. Ils contiennent des vecteurs de `Spot` avec des orientations spécifiques (Up, Down, Left, Right).
- **Détail technique - Conversion P2M** : Le fichier `Modules.cpp` utilise un helper statique **P2M (Pixel to Meter)**. Chaque pixel artistique est divisé par 7 (la constante `ART_PIXELS_PER_METER` du projet) pour garantir que toute la logique fonctionne sur une échelle physique cohérente.

Intégration de l'attachement et de la navigation Les modules sont assemblés à l'aide de vecteurs `AttachmentPoint`. Chaque point définit une coordonnée locale et un vecteur normal ; le `WorldGenerator` aligne l'attachement de sortie d'un module avec l'attachement d'entrée du suivant. La navigation est gérée en convertissant les données `LocalWaypoint` (relatives au centre du module) en positions `GlobalWaypoint` lors de l'instanciation de la scène.

III.2.c. Conteneur `World`

L'entité `World` (`World.hpp` / `World.cpp`) gère les contraintes visuelles et spatiales macroscopiques de la simulation.

Responsabilités :

- **Tiling de l'arrière-plan** : Génère et affiche un arrière-plan d'herbe procédural en utilisant une grille de textures de tuiles sélectionnées aléatoirement.
- **Application des contraintes spatiales** : Définit la largeur (`width`) et la hauteur (`height`) de la zone de simulation.
- **Rendu des superpositions (Overlays)** : Dessine la bordure noire globale et une grille de diagnostic optionnelle espacée de 1 mètre.
- **Le masque de premier plan (Foreground Mask)** : Implémente `drawMask()` , qui affiche de grands rectangles sombres en dehors de la zone jouable pour masquer les routes externes « infinies » et créer un viewport propre.

Détails d'implémentation technique :

- **Alignement des tuiles** : Les tuiles d'arrière-plan mesurent exactement `Config::BACKGROUND_TILE_SIZE` pixels artistiques. La largeur et la hauteur du monde sont arrondies à l'unité supérieure lors de la génération pour s'assurer qu'aucune tuile partielle n'est visible sur les bords.

III.2.d. Système de `Waypoint`

La structure `Waypoint` (`Waypoint.hpp`) est l'unité fondamentale de navigation utilisée par l'IA des agents.

Paramètres :

- **Position** : Coordonnées globales en mètres.
- **Tolerance** : Le rayon (en mètres) autour du point dans lequel une voiture doit entrer pour que le point soit considéré comme « atteint ».
- **Entry Angle** : Un angle d'entrée requis (en radians) que la voiture doit essayer de respecter lorsqu'elle atteint le point (crucial pour l'alignement sur les places de parking).
- **Stop At End** : Un flag booléen indiquant si l'agent doit marquer un arrêt complet à ce point.
- **Speed Limit Factor** : Un scalaire (de 0.0 à 1.0) qui limite la vitesse maximale du véhicule pour le segment se terminant à ce waypoint.

III.3. Simulation du trafic et règles globales

La gestion macroscopique de la simulation est assurée par la couche de coordination du trafic, qui régit les cycles de vie des agents et l'allocation des ressources.

III.3.a. Coordinateur `TrafficSystem`

Le `TrafficSystem` (`TrafficSystem.hpp` / `TrafficSystem.cpp`) agit comme le directeur de la simulation, gérant l'entrée, l'attribution des comportements et la suppression des agents.

Responsabilités :

- **Gestion de l'apparition (Spawn Management)** : Contrôle la fréquence d'entrée des véhicules en fonction d'un niveau d'apparition défini par l'utilisateur (allant de 0 à 5).
- **Surveillance globale** : Suit en permanence tous les agents actifs et leurs états respectifs (Conduite, Stationnement, Sortie).
- **Synchronisation des ressources** : Sert de pont entre l' `EntityManager` (qui détient les données) et la logique nécessaire pour manipuler ces données au fil du temps.

Détails d'implémentation technique :

- **Apparition à cadence fixe (Fixed-Cadence Spawning)** : Le système utilise un `spawnTimer` accumulé pendant l'événement `GameUpdateEvent` . Lorsque le minuteur dépasse l'intervalle défini dans `Config::Spawner::SPAWN_RATES` , une nouvelle entrée de véhicule est déclenchée.

- **Logique d'entrée** : Les véhicules apparaissent de manière aléatoire sur les segments de route situés soit à l'extrême gauche, soit à l'extrême droite. Le système calcule le décalage de voie approprié (**lane offset** : voie montante vs voie descendante) en fonction du côté d'entrée pour s'assurer que les véhicules circulent du bon côté de la chaussée.

Exemple de code : Logique de direction d'apparition et de voie

```cpp
// Déterminer l'ordonnée Y de la voie cible en fonction de la direction d'entrée
float laneY = entryLeft ? LANE_OFFSET_DOWN : LANE_OFFSET_UP;
Vector2 initialVel = { entryLeft ? initialSpeed : -initialSpeed, 0 };
auto spawnedCar = std::make_unique<Car>( {entryX, laneY}, worldPtr, initialVel, type);
```

III.3.b. Arbitrage de destination et heuristiques

Lorsqu'un véhicule entre dans le monde, le `TrafficSystem` effectue un processus de sélection logique pour lui assigner une destination.

Logique de sélection :

- **Filtrage** : Les véhicules sont d'abord filtrés par compatibilité. Les véhicules à combustion sont limités aux modules de parking. Les véhicules électriques (EV) évaluent leur niveau de batterie :
  - **Batterie < 30 %** : Obligation de chercher une station de recharge (`Charging Station`).
  - **Batterie > 70 %** : Obligation de chercher une place de parking (`Parking Spot`).
  - **Entre 30 % et 70 %** : Un tirage aléatoire pondéré détermine si l'agent cherche à se recharger ou à se garer.
- **Heuristiques de priorité** :
  - `PRIORITY_DISTANCE` : Le système recherche l'installation compatible la plus proche disposant d'au moins une place libre.
  - `PRIORITY_PRICE` : Le système échantillonne une place libre aléatoire dans chaque installation compatible et sélectionne l'installation offrant le prix absolu le plus bas.
- **Solution de repli pour le trafic de transit (Through-Traffic Fallback)** : Si aucune installation compatible n'a de place libre, l'agent se voit attribuer un chemin de « passage direct » qui le mène directement à l'autre côté de la carte sans s'arrêter.

III.3.c. Gestion de l'occupation et des places (Spots)

La gestion des ressources est traitée via un modèle de transition à trois états pour chaque place individuelle : `FREE` (Libre), `RESERVED` (Réservée) et `OCCUPIED` (Occupée).

Transitions d'état :

1. **Réservation** : Déclenchée immédiatement lors de la sélection de la destination. Cela empêche deux voitures de revendiquer la même place pendant qu'elles sont en transit.
2. **Occupation** : Transition effectuée par le `TrafficSystem` lorsque l'état de l'agent `Car` passe à `PARKED` ou lorsqu'il atteint le waypoint d'alignement à l'entrée de l'installation.
3. **Libération (Release)** : Se produit lorsque le véhicule décide de quitter l'installation, rendant la place immédiatement disponible pour de nouvelles réservations.

III.3.d. Surveillance du cycle de vie des agents

Le `TrafficSystem` surveille l'état interne de chaque véhicule pour gérer les transitions entre les différentes phases de la simulation.

Étapes du cycle de vie :

- **La session de stationnement** :
  - **Véhicules génériques** : Restent garés pendant une durée aléatoire définie par un minuteur local.
  - **Véhicules électriques (EV)** : Se rechargent à un taux défini dans `Config::CHARGING_RATE`. Ils restent dans l'installation jusqu'à ce qu'ils atteignent le seuil `BATTERY_EXIT_THRESHOLD` (80 %), point à partir duquel ils ont une probabilité de partir, probabilité qui devient une certitude au seuil `BATTERY_FORCE_EXIT_THRESHOLD` (95 %).

- **Sortie** : Lorsqu'une session se termine, le système libère la place et utilise le `PathPlanner` pour générer une trajectoire de sortie afin de réintégrer le réseau routier.
- **Suppression (Despawning)** : Le système identifie les véhicules qui ont atteint les limites du monde et les supprime de l' `EntityManager`, garantissant ainsi la récupération de la mémoire.

III.3.e. Système de suivi de caméra

Le `TrackingSystem` (`TrackingSystem.hpp` / `TrackingSystem.cpp`) fournit un service de gestion automatisée de la vue (viewport), permettant à l'utilisateur de suivre des agents spécifiques à travers l'environnement de la simulation.

Responsabilités :

- **Ciblage d'agent** : Répond à l'événement `StartTrackingEvent` pour commencer la surveillance d'une instance spécifique de `Car`.
- **Synchronisation de la vue** : Calcule la position actuelle de la cible à chaque image (*frame*) et publie les données `CameraMoveEvent` vers le `CameraSystem`.
- **Gestion automatique du cycle de vie** : Détecte lorsqu'un agent suivi quitte la simulation et met fin automatiquement à la session de suivi via un `StopTrackingEvent`.

Détails d'implémentation technique :

- **Observation découplée** : Le système ne détient pas de référence forte vers l'objet `Car` ; il surveille l'état de l'agent (par exemple, `EXITING`) pour déterminer le moment opportun pour se désengager en toute sécurité.
- **Contrôle piloté par les événements** : En publiant des événements de mouvement au lieu de manipuler directement la caméra, le système maintient une séparation nette entre la logique de simulation et le rendu de la vue.

III.4. IA des agents autonomes et navigation

L'intelligence de la simulation réside dans les agents `Car` et l'utilitaire `PathPlanner`, qui collaborent pour simuler des comportements de conduite complexes.

III.4.a. Comportements de direction (Steering Behaviors) et physique

Chaque entité `Car` (`Car.hpp` / `Car.cpp`) utilise un modèle de direction de style Reynolds (**Reynolds-style steering model**) pour naviguer dans le monde.

Mécaniques de base :

- **Comportement Seek** : L'agent calcule une « vitesse désirée » (**desired velocity**) vers le waypoint actuel. La « force de direction » (**steering force**) est la différence entre cette vitesse désirée et la vitesse actuelle, limitée (**clamped**) par la valeur `maxForce`.
- **Intégration de la vélocité** : La vélocité est mise à jour par l'accélération (force/masse) à chaque frame. La position de l'agent est ensuite mise à jour par la vélocité, ce qui produit un mouvement fluide basé sur le momentum (inertie).
- **Limites de vitesse dynamiques** : Chaque waypoint peut spécifier un `speedLimitFactor`. La voiture (`Car`) module automatiquement sa vitesse cible en fonction du segment qu'elle traverse (par exemple, en ralentissant dans les zones de parking).
- **Ralentissement en virage (Turn Slowdown)** : Pour éviter un effet de « dérive » (**drifting**) peu naturel, l'agent calcule l'angle entre son orientation actuelle (**heading**) et l'orientation du waypoint cible. Si l'angle est aigu et la distance proche, l'agent effectue une interpolation linéaire de sa vitesse vers une « vitesse de virage sûre » définie dans `Config::CarAI::TURN_MIN_SPEED_FACTOR`.

Exemple de code : Implémentation du Steering

```cpp
// Le comportement 'Seek' de base calcule une force de direction vers la cible
Vector2 desired = Vector2Scale(Vector2Normalize(Vector2Subtract(target, pos)), speed);
Vector2 steer = Vector2Subtract(desired, velocity);
applyForce(Vector2ClampValue(steer, 0, maxForce));
```

III.4.b. Évitement de collision et logique Stop-Start

Les agents possèdent une conscience spatiale leur permettant de naviguer en toute sécurité dans des scénarios à fort trafic.

Détails d'implémentation technique :

- **Corridor de détection** : À chaque frame, l'agent projette un rectangle virtuel (corridor) devant sa position. Il vérifie la présence d'autres agents à l'intérieur d'une distance de visibilité (**lookAheadDist**) qui varie proportionnellement à la vitesse actuelle du véhicule.
- **Freinage adaptatif (Adaptive Braking)** : si un autre agent est détecté dans le corridor, le véhicule applique une contre-force proportionnelle au carré de la proximité, simulant un comportement de freinage naturel.
- **Amortissement critique (Hard Stop)** : si la distance tombe en dessous d'un seuil critique (par exemple, 3,2 m), l'agent force l'amortissement de sa vitesse à 0, garantissant qu'aucun chevauchement physique ne se produise.
- **Séparation latérale** : les agents appliquent une force répulsive de faible intensité aux voisins situés à moins de 1,9 m pour simuler le centrage dans la voie et éviter les collisions latérales (**side-swiping**) lors des virages.
- **Atténuation des blocages (Deadlock Mitigation)** : si la vitesse d'un agent reste proche de zéro alors qu'il n'est pas dans l'état `PARKED`, il applique une impulsion latérale périodique (un « **wiggle** ») pour contourner les obstructions.

III.4.c. `PathPlanner` et trajectoires géométriques

Le `PathPlanner` (`PathPlanner.hpp` / `PathPlanner.cpp`) est un moteur de géométrie statique qui traduit la connectivité modulaire en une séquence de waypoints traversables.

Assemblage de la trajectoire :

- **Entrée de route (Road Entry)** : Calcule le point de transition optimal sur la route principale en fonction de l'orientation de l'installation (Up/Down).
- **Entrée de porte (Gate Entry)** : Génère un point à la porte de l'installation, en s'assurant que le véhicule entre à une « profondeur de porte » (**Gate Depth**) spécifique par module (définie dans `Config::GateDepth`) pour dégager la route d'accès.
- **Point d'alignement (Alignment Point)** : Calcule une position d'arrêt devant la place cible, garantissant que le véhicule est parallèle à la place avant de tenter de se garer.
- **Finalité de la place (Spot Finality)** : le dernier waypoint est le centre de la place de parking ou de recharge elle-même, marqué avec le flag `stopAtEnd` et l'angle d'entrée requis (`entryAngle`).

Processus d'assemblage de trajectoire Le `PathPlanner` assemble un chemin en raccordant des segments discrets : 1. **Segment d'entrée (Entry Segment)** : Du point d'apparition jusqu'au connecteur de la route d'entrée. 2. **Segment d'accès (Access Segment)** : Navigation de la route vers la voie interne de l'installation. 3. **Segment de manœuvre (Maneuver Segment)** : Traversée de la voie interne jusqu'au point d'alignement de la place. 4. **Segment d'amarrage (Docking Segment)** : Alignement final et transition vers l'état `PARKED`.

III.4.d. Phases de l'IA et modes comportementaux

La simulation utilise des « Phases comportementales » pour ajuster les performances des agents en fonction de leur environnement.

Phases définies (`Config::Phases`) :

- **HIGHWAY** : (Vitesse 100 %, Tolérance 10 m). Utilisée pour les trajets sur la route principale ; permet un mouvement fluide et rapide.
- **APPROACH** : (Vitesse 100 %, Tolérance 3 m). Utilisée lors de l'approche d'une entrée d'installation spécifique.
- **ACCESS** : (Vitesse 40 %, Tolérance 4 m). Utilisée pour la conduite sur les routes d'accès aux installations.
- **MANEUVER** : (Vitesse 20 %, Tolérance 2 m). Utilisée pour la navigation intérieure des parkings.
- **PARKING** : (Vitesse 10 %, Tolérance 0,3 m). Mode de haute précision utilisé pour l'alignement final sur la place et l'amarrage (**docking**).

III.5. Éléments d'interface utilisateur (UI) et interaction

La couche d'interface utilisateur fournit les commandes et les mécanismes de retour d'information nécessaires pour configurer et analyser la simulation.

III.5.a. `UIElement` de base et `UIManager`

Le système d'UI est construit sur une architecture de composants modulaires (`UIElement.hpp` / `UIManager.hpp`).

Responsabilités :

- **Gestion du cycle de vie** : L' `UIManager` gère un conteneur d'objets `UIElement` , déléguant les appels `update()` et `draw()` à tous les composants actifs.
- **Découplage des composants** : Les éléments d'interface communiquent avec le back-end de la simulation exclusivement via l' `EventBus` .
- **Persistance de l'état** : Chaque élément suit sa propre visibilité et son état « actif », permettant des basculements dynamiques du HUD.

Composants standard :

- **`UIButton`** : Un composant polyvalent qui gère les collisions à la souris (survol/pression) et déclenche un rappel (*callback*) `std::function` configurable lors du relâchement. Il utilise des palettes de couleurs néon standardisées pour assurer une cohérence visuelle.
- **`MuteButton`** : Un bouton global spécialisé, géré par l' `AudioManager` , qui bascule l'état audio en publiant des commandes de sourdine internes et en mettant à jour les icônes de volume interactives.

III.5.b. Tableau de bord (Dashboard) et sélection de véhicule

Le `DashboardOverlay` (`DashboardOverlay.hpp` / `DashboardOverlay.cpp`) est l'outil de diagnostic principal de la simulation.

Fonctionnalités interactives :

- **Informations contextuelles** : Le tableau de bord écoute l'événement `EntitySelectedEvent` et adapte son contenu d'affichage :
  - **Mode général** : Affiche le temps d'exécution global (uptime), le FPS et le nombre total de véhicules.
  - **Mode Voiture** : Affiche le type de propulsion du véhicule spécifique, le pourcentage de batterie, l'état actuel de l'IA et l'heuristique de priorité.
  - **Mode Installation** : Affiche les pourcentages d'occupation, le nombre total de places et les prix dynamiques des places.
- **Logique de sélection** : Dans `GameScene.cpp` , les clics de souris sont convertis de l'espace écran (Screen Space) vers l'espace monde (World Space en mètres) à l'aide du `CameraSystem` . La scène effectue ensuite des tests de proximité par rapport aux hitboxes de toutes les entités pour déterminer la cible de l'événement de sélection.

Interaction HUD pilotée par les événements Le Dashboard réagit aux événements globaux pour basculer sa visibilité ou rafraîchir sa source de données.

```cpp
// Les éléments UI réagissent aux événements sans connaître l'état de la simulation
bus->subscribe<EntitySelectedEvent>([this](const auto& e) {
    this->currentSelection = e;
    this->visible = true; // Afficher automatiquement le HUD lors de la sélection
});
```

III.5.c. Logique spécifique aux scènes

L'état de l'application est partitionné en « Scènes » isolées, chacune héritant de `IScene` .

Scènes principales :

- **`MainMenuScene`** : Le point d'entrée, permettant la navigation vers le menu de configuration ou la sortie de l'application.
- **`MapConfigScene`** : Une interface spécialisée où les utilisateurs peuvent ajuster les paramètres de la carte générée procéduralement (par exemple, le nombre et le type d'installations).
- **`GameScene`** : L'étape principale de la simulation. Elle initialise tous les systèmes de jeu centraux (`TrafficSystem` , `EntityManager` , `CameraSystem`) et gère le mapping des entrées de haut niveau pour la pause et la manipulation de la caméra.

III.5.d. Contrôles interactifs de vitesse et d'audio

L'utilisateur garde le contrôle sur le flux de la simulation grâce à des commandes en temps réel.

Interactions clés :

- **Multiplicateur de vitesse** : Les utilisateurs peuvent faire défiler la vitesse de simulation (de 1x à 3x) via les boutons du HUD. Cela publie un `SimulationSpeedChangedEvent` , que la `GameLoop` utilise pour ajuster le taux de consommation de l'accumulateur temporel.
- **Manipulation de la caméra** : `GameScene` capture les entrées de défilement (scroll) et de glissement (drag) de la souris pour publier des événements de caméra, permettant aux utilisateurs de zoomer et de se déplacer (pan) à travers le monde généré.
- **Pause/Reprise** : L'état global de la simulation peut être basculé via la touche `P` ou les boutons du HUD, ce qui suspend la publication de `GameUpdateEvent` tout en maintenant le pipeline de rendu actif pour l'interaction avec l'interface utilisateur.

III.6. Vérification et Infrastructure

La fiabilité et la portabilité du projet sont assurées par une infrastructure de build et de test robuste.

III.6.a. Tests unitaires avec GoogleTest

La base de code comprend une suite de tests complète (répertoire `tests/`) s'appuyant sur le framework **GoogleTest** afin de garantir la correction au niveau des composants.

Modules de tests principaux :

- **`EventBusTests.cpp`** : Valide la distribution d'événements type-safe, la gestion du cycle de vie des abonnements (RAII) et la publication thread-safe.
- **`GameLoopTests.cpp`** : Vérifie la logique de l'accumulateur temporel, garantissant que les mises à jour logiques se produisent exactement au pas de temps fixe (**fixed delta time**), indépendamment de la variabilité des frames.
- **`SceneManagerTests.cpp`** : Confirme que les transitions de scène déclenchent correctement les séquences `load()` / `unload()` et qu'une seule scène est active à la fois.
- **`WindowTests.cpp`** : Teste les mathématiques de mise à l'échelle des coordonnées logiques vers physiques et le calcul du ratio d'aspect.
- **`GameSceneTests.cpp`** : Fournit des tests d'intégration de haut niveau, vérifiant que les événements de génération du monde sont publiés et que les systèmes sont initialisés lors de l'entrée en scène.
- **`MainTests.cpp`** : Valide la logique de base de la simulation, y compris le comportement des batteries des voitures, les heuristiques d'arbitrage des destinations et la précision de la planification de trajectoire. Il s'appuie sur le modèle centralisé `AssetManager::LoadAllAssets()` pour garantir une configuration d'environnement cohérente lors de la vérification.

III.6.b. Système de build modulaire CMake

Le fichier `CMakeLists.txt` est conçu pour une compilation moderne et multiplateforme avec une gestion des dépendances totalement automatisée.

Caractéristiques techniques :

- **Acquisition automatisée des dépendances** : Utilise `FetchContent` pour télécharger et compiler Raylib 5.5 directement depuis la source officielle, garantissant un environnement cohérent sur les différentes machines de développement.
- **Portabilité des Assets** : Implémente une commande personnalisée `POST_BUILD` qui synchronise automatiquement le répertoire `assets` avec le répertoire binaire de l'exécutable, évitant les erreurs d'exécution de type « fichier non trouvé ».
- **Respect des standards** : Impose explicitement le standard C++20 et active les avertissements du compilateur de haute sévérité (`-Wall -Wextra -Wpedantic`) pour détecter les bugs potentiels dès la compilation.

III.6.c. Sécurité mémoire et nettoyage des ressources

Le projet respecte les principes **RAII** (*Resource Acquisition Is Initialization*) pour tous les sous-systèmes critiques.

Modèles d'implémentation :

- **Smart Pointers** : Utilise `std::unique_ptr` pour la propriété exclusive (par exemple, au sein de l' `EntityManager`) et `std::shared_ptr` pour les ressources partagées (par exemple, l' `EventBus`).
- **Subscription Tokens** : Les écouteurs sur l' `EventBus` sont gérés via des objets `Subscription` . Lorsque ces objets sortent du scope, le destructeur désinscrit automatiquement l'écouteur du bus, empêchant les pointeurs pendants (*dangling pointers*) et les rappels fantômes (*ghost callbacks*).
- **Déchargement des Assets** : L' `AssetManager` fournit une méthode centralisée `UnloadAll()` appelée lors de la fermeture de l'application pour libérer explicitement les textures GPU, garantissant l'absence de fuites mémoire.

III.6.d. Standards de documentation du code

La transparence technique est maintenue grâce à un style de commentaire strict adapté aux générateurs de documentation comme **Doxygen**.

Exigences :

- **Documentation In-Header** : Toutes les classes et méthodes publiques doivent inclure les balises `@brief` , `@param` , et `@return` dans leurs fichiers d'en-tête.
- **Récit d'implémentation** : Les algorithmes critiques (par exemple, la boucle « Fix Your Timestep » ou la logique de placement du `WorldGenerator`) incluent des commentaires inline expliquant le raisonnement mathématique derrière chaque étape spécifique.

IV. Expérience utilisateur et présentation des fonctionnalités

Cette section propose un parcours chronologique de l'application ParkLogic du point de vue de l'utilisateur, en détaillant les fonctionnalités interactives et le cycle de vie de la simulation.

IV.1. Phase I : Entrée et navigation initiale

Dès le lancement de ParkLogic, l'utilisateur est accueilli par la **MainMenuScene**. Cette interface sert de portail à l'application, offrant un style visuel épuré en vue de dessus, cohérent avec l'environnement de la simulation.

- **Start Simulation** : Dirige l'utilisateur vers l'étape de configuration.
- **Exit** : Ferme proprement l'application, garantissant que toutes les ressources GPU et les flux audio sont libérés.

IV.2. Phase II : Configuration procédurale

Avant que la simulation ne commence, l'utilisateur accède à la **MapConfigScene**. Cette étape permet de personnaliser le « Plan du monde » que le `WorldGenerator` finira par exécuter.

- **Équilibrage des installations** : Les utilisateurs peuvent augmenter ou diminuer le nombre de **Parkings** (Parking Facilities) et de **Stations de recharge EV** (EV Charging Stations).
- **Retour interactif** : L'interface utilisateur empêche les configurations invalides (par exemple, zéro installation) et affiche en temps réel le décompte de l'infrastructure demandée.
- **Déclenchement de la génération** : En cliquant sur « Generate World », l'utilisateur publie un `GenerateWorldEvent`, faisant passer l'application à l'étape principale de simulation.

IV.3. Phase III : Engagement dans la simulation

La **GameScene** est le cœur de l'application, là où l'utilisateur observe et interagit avec les agents autonomes.

IV.3.a. Conscience spatiale et contrôle de la caméra

L'utilisateur dispose d'une liberté totale sur le viewport, permettant aussi bien une observation macroscopique qu'une analyse microscopique du comportement des agents.

- **Panoramique (Panning)** : Maintenir le bouton droit de la souris (ou utiliser les touches directionnelles) permet à l'utilisateur de faire glisser la caméra à travers le monde généré.
- **Zoom** : L'utilisation de la molette de la souris modifie dynamiquement l'échelle de la vue, permettant à l'utilisateur de zoomer pour observer la précision du stationnement ou de dézoomer pour voir l'ensemble du flux de trafic.
- **Limitation (Clamping)** : Le système de caméra impose automatiquement des limites, empêchant l'utilisateur de glisser dans le « vide » au-delà du monde généré.

IV.3.b. Découverte d'informations et tableau de bord (Dashboard)

L'interaction est principalement pilotée par la sélection. En faisant un clic gauche sur n'importe quelle voiture ou installation, l'utilisateur « focalise » son attention sur cette entité.

- **Mise en évidence de l'entité** : Les voitures sélectionnées affichent leur chemin cible sous la forme d'une série de segments de ligne connectés, fournissant une représentation visuelle de l'intention de l'IA.
- **Le HUD du Dashboard** : Une superposition contextuelle apparaît dans le coin de l'écran :
  - **Statistiques générales** : Affiche le nombre total de véhicules, le temps d'exécution de la simulation et le FPS.
  - **Analyse du véhicule** : Affiche le type de propulsion de la voiture, son niveau de batterie actuel, sa phase d'IA (ex: « Maneuvering ») et son heuristique de destination active (Priorité : Prix vs Distance).
  - **Analyse de l'installation** : Affiche les taux d'occupation et les prix actuels des places.

IV.3.c. Contrôle temporel et environnemental

L'utilisateur peut moduler le « pouls » de la simulation grâce à plusieurs commandes en temps réel :

- **Vitesse de simulation** : Un bouton dédié fait défiler la vitesse de simulation de **1x** jusqu'à **3x**. Cela permet aux utilisateurs d'accélérer les sessions de stationnement de longue durée ou de ralentir l'action pour déboguer des scénarios complexes d'évitement de collision.
- **Pause/Reprise** : L'utilisateur peut « geler » la logique de simulation à tout moment (via la touche `P` ou le bouton du HUD), permettant une analyse statique de l'état actuel du monde tout en gardant l'interface utilisateur et les systèmes de caméra actifs.
- **Bascule audio** : Un bouton de sourdine globale, géré par l'`AudioManager`, permet à l'utilisateur d'activer ou de désactiver la musique d'ambiance et les effets sonores. L'état du bouton est conservé en parallèle du contrôle du volume global.

IV.3.d. Interaction active : Apparition de voitures (Car Spawning)

Bien que le système gère l'apparition autonome par défaut, l'utilisateur peut prendre le contrôle direct de la population de la simulation.

- **Apparition manuelle** : En cliquant sur le bouton « Spawn Car » dans le HUD, l'utilisateur déclenche manuellement le `CreateCarEvent`, injectant un nouvel agent dans la simulation à un point d'entrée aléatoire.

- **Ajustement du niveau d'apparition** : Les utilisateurs peuvent faire défiler les « Niveaux de Spawn » (de 0 à 5), ce qui ajuste dynamiquement la fréquence de la logique d'apparition automatisée du `TrafficSystem`. Le niveau 0 arrête efficacement les nouvelles entrées, tandis que le niveau 5 crée une densité de trafic maximale.

IV.4. Phase IV : Le cycle de simulation

ParkLogic est conçu pour l'expérimentation itérative.

- **Réinitialisation du monde** : À tout moment, l'utilisateur peut appuyer sur la touche `Echap` pour revenir au menu principal. De là, il peut accéder à nouveau à l'étape de configuration, ajuster les paramètres et générer une disposition de carte complètement différente.
- **Persistance de l'apprentissage** : En observant comment différentes configurations de cartes gèrent une densité de trafic de niveau 5, les utilisateurs peuvent obtenir des informations sur l'efficacité de diverses dispositions routes-installations et sur les heuristiques de tarification.

V. Rétrospective et raffinements architecturaux

En revenant sur le développement de ParkLogic, plusieurs décisions fondamentales pourraient être affinées si le projet était initié de nouveau. Ces raffinements se concentrent sur le renforcement de la base technique, l'amélioration des performances via une conception orientée données (data-oriented design) et l'élimination des dépendances codées en dur.

V.1. Adoption d'une architecture orientée données (EnTT)

Bien que la hiérarchie polymorphe actuelle offre une clarté conceptuelle, une base supérieure impliquerait une approche strictement **Data-Oriented Design (DOD)** propulsée par une bibliothèque comme **EnTT**. L'utilisation d'un ECS (Entity Component System) mature dès le départ permettrait de résoudre plusieurs limitations structurelles :

- **Élimination de l'overhead du dispatch virtuel** : En remplaçant l'héritage par des pools de composants plats, le CPU éviterait le surcoût des recherches dans les tables virtuelles (vtable lookups) lors des mises à jour massives des agents de voitures et des modules du monde.
- **Optimisation de la localité du cache (Cache Locality)** : Les composants seraient stockés de manière contiguë en mémoire, réduisant considérablement les défauts de cache (cache misses) et permettant à la simulation de gérer des milliers d'entités sans dégradation linéaire des performances.
- **Décomposition de la logique** : La logique serait entièrement déplacée dans des systèmes autonomes opérant sur des ensembles de données minimaux, empêchant la tendance « God Object » observée dans certains coordinateurs actuels.

V.2. Dispatching d'événements natif (`entt::dispatcher`)

Dans un scénario de redéveloppement, l'`EventBus` personnalisé serait remplacé par une solution plus performante et intégrée, spécifiquement l'**EnTT Dispatcher**.

- **Intégration étroite du système** : L'utilisation du dispatcher natif permet une communication fluide entre les systèmes ECS et la logique pilotée par les événements, garantissant que les changements d'état (comme une voiture entrant sur une place) sont traités avec une latence minimale.
- **Efficacité mémoire** : L'exploitation de l'infrastructure de messagerie interne de la bibliothèque éliminerait le besoin de wrappers de type-erasure personnalisés et de maps d'abonnés gérées manuellement, réduisant ainsi l'empreinte mémoire du moteur.
- **Pipeline déterministe** : Un dispatcher natif facilite un contrôle plus clair sur l'ordre d'exécution des événements au sein de la boucle fixed-timestep, aidant au débogage de conditions de concurrence (race conditions) complexes.

V.3. Framework moteur-simulation découplé

L'architecture actuelle du projet présente un certain couplage entre le code spécifique à la simulation (ex: `TrafficSystem`) et les services du moteur de base (ex: `Window`). Recommencer impliquerait de construire une frontière **Engine-Simulation** plus nette :

- **Cœur de moteur autonome (Standalone Engine Core)** : La gestion des fenêtres, le polling des entrées, la mise en cache des ressources (Asset Manager) et les systèmes audio seraient extraits dans une bibliothèque moteur agnostique au projet.
- **La simulation comme Plugin** : La génération du monde, l'IA des agents et les règles économiques seraient implémentées comme une couche de simulation spécialisée reposant sur le cœur du moteur, communiquant uniquement via des interfaces standard.

V.4. Configuration et schémas pilotés par les données (Data-Driven)

Pour éviter la rigidité des constantes codées en dur dans `Config.hpp`, un remake donnerait la priorité à une architecture entièrement **data-driven** :

- **Spécification externe** : Tous les paramètres de simulation (ex: vitesses des véhicules, multiplicateurs de prix des installations, seuils de la logique d'apparition) seraient définis dans des fichiers externes **JSON ou YAML**.
- **Chargement validé par schéma** : L'application utiliserait une validation par schéma au démarrage pour s'assurer que les fichiers de configuration externes sont structurellement corrects, évitant les crashs à l'exécution dus à des données malformées.
- **Itération rapide** : Cette approche permettrait d'ajuster l'équilibre de la simulation et les règles de génération du monde sans nécessiter de recompilations incrémentales du binaire C++.

V.5. Abstraction intrinsèque du pipeline de rendu

Au lieu d'appels directs à Raylib entremêlés avec la logique du monde, une architecture affinée utiliserait une **Render Pipeline Abstraction** :

- **Support natif des Shaders** : Le moteur de rendu serait conçu dès la première itération pour supporter les shaders GLSL pour des diagnostics visuels avancés (ex: heatmaps de terrain) et des effets esthétiques (ex: anti-aliasing) sans modifier le code des entités.
- **Rendu basé sur l'état (State-Based Rendering)** : Les draw calls seraient regroupés (batched) et triés en fonction du matériau ou de l'ordre Z (Z-order) avant d'être soumis au GPU, maximisant l'efficacité de l'API graphique sous-jacente.
- **Logique HUD détachée** : Le rendu de l'interface utilisateur serait entièrement séparé de la vue du monde, garantissant que les superpositions de diagnostic de la simulation n'interfèrent pas avec les calculs de rendu spatial.

VI. Conclusion et bilan

Le projet ParkLogic représente une exploration complète du développement de simulations d'agents autonomes en utilisant le C++20 moderne et la bibliothèque graphique Raylib. Partant d'une page blanche, cet effort a abouti à un environnement fonctionnel où la génération procédurale du monde et les heuristiques de trafic basées sur les agents opèrent dans une boucle synchronisée et pilotée par les événements.

VI.1. Audit des Design Patterns architecturaux

L'implémentation technique repose sur plusieurs modèles de conception logicielle établis pour gérer la complexité et maintenir la séparation des préoccupations :

- **Observer Pattern** : Utilisé via l'`EventBus` global pour faciliter une communication découplée entre les systèmes (par exemple, le `TrafficSystem` réagissant au `GameUpdateEvent`).

- **State Pattern** : Implémenté à la fois dans le `SceneManager` pour piloter le flux de l'application à haut niveau et au sein des agents `Car` pour gérer les phases de mouvement (`DRIVING` , `PARKED` , `EXITING`).
- **Strategy & Heuristics** : Le `TrafficSystem` emploie des heuristiques de type stratégie (`PRIORITY_PRICE` vs. `PRIORITY_DISTANCE`) pour dicter de manière autonome le comportement des agents lors de l'arbitrage de destination.
- **RAII (Resource Acquisition Is Initialization)** : Strictement appliqué grâce à l'utilisation de smart pointers (`std::unique_ptr` , `std::shared_ptr`) pour la propriété des sous-systèmes et des tokens `Subscription` pour le désabonnement automatique aux événements.
- **Owner-Subsystem Pattern** : La classe `Application` agit comme coordinateur central, gérant le cycle de vie et l'accès partagé aux sous-systèmes utilitaires de base.

VI.2. Évaluation technique objective

Bien que le projet démontre une intégration réussie de systèmes divers — allant de l'interpolation physique à la géométrie procédurale — il doit être évalué objectivement en tant que produit logiciel.

D'un point de vue de l'ingénierie, le projet manque d'une fondation architecturale rigoureuse, telle qu'un système formel d'Entity Component System (ECS). L'architecture qui en résulte se caractérise par un assemblage de systèmes disjoints, maintenus ensemble par des ponts ad hoc fragiles qui privilégient la fonctionnalité immédiate au détriment de l'intégrité systémique. Ce manque de fondation structurelle formelle a produit une base de code chargée d'une dette technique significative, limitant sévèrement sa maintenabilité à long terme.

De plus, bien que techniquement impressionnant en tant que première incursion dans le développement de systèmes en C++, ParkLogic manque de la profondeur, de l'équilibre et du polissage mécanique requis pour être classé comme un jeu réussi. Il fonctionne efficacement comme un bac à sable de simulation (simulation sandbox) mais offre une valeur ludique ou une profondeur centrée sur le joueur limitée.

VI.3. Déclaration finale

En conclusion, ParkLogic a servi d'outil pédagogique inestimable pour maîtriser les complexités de la simulation en temps réel, de la gestion de la mémoire et de l'architecture pilotée par les événements. Le projet témoigne de la puissance de la génération procédurale et des défis de la coordination d'agents autonomes. Les leçons tirées de ces lacunes structurelles fournissent un plan d'action clair pour des approches plus rigoureuses et orientées données (data-oriented) qui définiraient tout futur effort d'ingénierie dans ce domaine.